UNCLASSIFIED

(4)

Information Technology Division

[redacted]

[redacted]

UNCLASSIFIED

4

UNCLASSIFIED

AR-006-402

DSTO AUSTRALIA

ELECTRONICS RESEARCH LABORATORY

Information Technology Division

TECHNICAL REPORT
ERL-0512-TR

GUMNUT SPECIFICATION AND REPORT

by

Richard A. Altmann, Michael A. Fitzgerald and Peter S. Keays

SUMMARY

Gumnut is a part of MultiView, an integrated programming environment. By means of a number of tools operating possibly concurrently, over a distributed workstation network, MultiView supports the development of software in a growing number of programming languages. Gumnut and its associated meta-language (or language to describe a language) is the tool which allows MultiView to be extended for a new programming language. This report describes Gumnut and the meta-language.

MAY 1990

COPY No. [illegible]

APPROVED FOR PUBLIC RELEASE

D

POSTAL ADDRESS: Director, Electronics Research Laboratory, PO Box 1600, Salisbury, South Australia, 5108.

UNCLASSIFIED

Contents

List of Figures

List of Tables

1 Introduction

1.1 Purpose

This paper gives the specification of the Gumnut translator, a syntax driven tool used in conjunction with the MultiView integrated programming environment[3]. In Chapter 2 we discuss the fundamentals of Abstract Syntax on which the meta-language used by Gumnut is based. In Chapter 3 the meta-language is specified. Chapter 4 specifies the Gumnut translation processing, while Chapter 5 specifies the output, by the use of a small example. In Chapter 6 we report on Scanners developed for the Ada programming language. Chapter 7 proposes extensions to the Gumnut translator and appendices give an example, a complete Backus-Naur Form (BNF) definition of the meta-language, and a list of input scanners implemented at the time of writing. The remainder of this chapter will give a brief history surrounding the need for the Gumnut translator.

1.2 Gumnut

Gumnut is a translator which assists in instantiating the MultiView distributed integrated incremental programming environment for a particular language. It was originally designed in April 1986 by Michael M^{c}Carthy, at the University of Adelaide, but has since been extended by the authors at Information Technology Division (ITD), ERL, DSTO, Salisbury.

MultiView is a distributed incremental integrated programming environment which has been under development at the University of Adelaide since 1985, under the direction of Dr. Chris Marlin. For an overview of MultiView, see [7] or [3].

MultiView was developed so that its generic components could be instantiated via a syntactic description of the programming language to be supported. MultiView must ensure that only syntactically correct programs are created within the environment so information is required about the structure of each supported programming language; *abstract syntax* was chosen as a way of representing the structure of a language. Gumnut was then developed to take an abstract syntax description of a language and create the necessary data files which would then be used by the components within MultiView. For example, one data file contains menus, for use by template-driven editors. Figure 1 gives a pictorial view of how Gumnut is used to instantiate MultiView. A description of the language is passed to the Gumnut translator, which outputs language specific data, which is then encapsulated into MultiView, to form a programming environment for the language described.

Gumnut was extended at ITD to allow for easier entry of the abstract syntax and also to provide additional outputs to implement scanning, of such entities as numbers, strings and identifiers.

1.3 Scope

The specification of output from Gumnut within this report is for use with the MultiView version 1. The meta-language described will remain the same, when Gumnut is able to be used with MultiView version 2, which will be implemented in Ada rather than Modula-2.

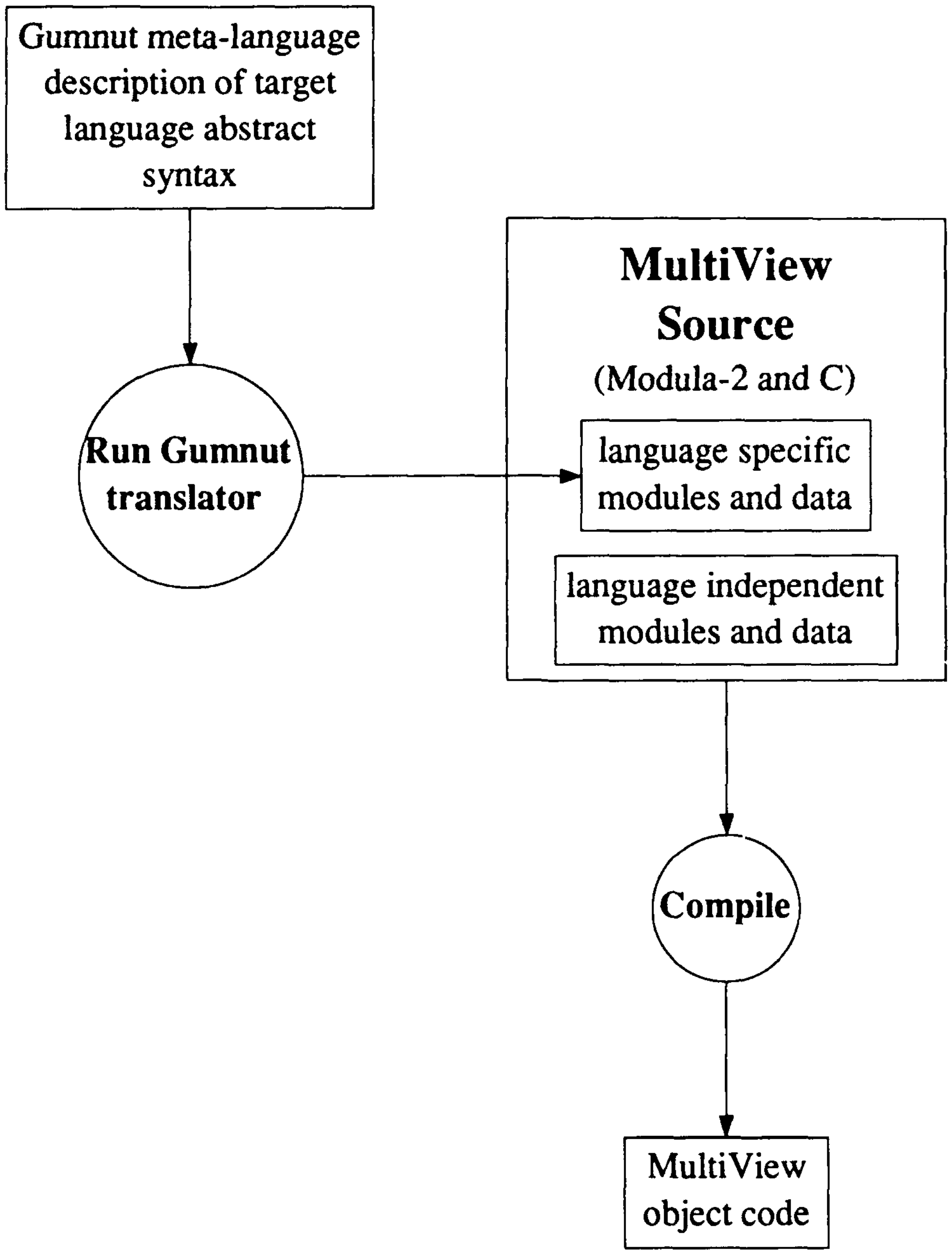

Gumnut meta-language description of target language abstract syntax
Run Gumnut translator
MultiView Source
(Modula-2 and C)
language specific modules and data
language independent modules and data
Compile
MultiView object code

Figure 1 Using Gumnut to instantiate MultiView

2 Abstract Syntax

2.1 Definitions

This chapter outlines how an abstract syntax [1] describes a programming language. An abstract syntax is the syntax of the language, with the components abstracted out. Thus it contains only syntactic categories, such as statements and declarations, as well as the structure of syntactic structures, such as the fact that a repeat loop contains a statement sequence and an expression controlling the termination of the loop. So all the "syntactic baggage", specifically keywords and punctuation, is removed.

The abstract syntax of a language, as used by M^c^Carthy [8], can be described using the following sets:

$\mathcal{U}$, the universe of symbols
$\mathcal{P} \subset \mathcal{U}$, the phyla (syntactic categories)
$\mathcal{F} \subset \mathcal{U}$, the fixed arity operators (syntactic structures with a fixed number of heterogeneous components)
$\mathcal{V} \subset \mathcal{U}$, the variable arity operators (syntactic structures with a variable number of homogeneous components)

such that $\mathcal{P}$, $\mathcal{F}$ and $\mathcal{V}$ are disjoint. Hence $\mathcal{P} \cup \mathcal{F} \cup \mathcal{V} = \mathcal{U}$. The functions which are used to manipulate the abstract syntax are:

$\pi{:}\mathcal{P} \rightarrow \rho(\mathcal{F} \cup \mathcal{V})$ (the structures within each syntactic category)
$\sigma{:}\mathcal{F} \rightarrow \mathcal{P}^*$ (the various components of a syntactic structure)
$\tau{:}\mathcal{V} \rightarrow \mathcal{P}$ (the syntactic category for components of a list)

where $\rho(x)$ is the powerset of x, and $\mathcal{P}^*$ is the set of ordered tuples of elements of $\mathcal{P}$, that is, $\{(x_1, x_2, \ldots, x_n) : x_1, x_2, \ldots, x_n \in \mathcal{P}, n \geq 0\}$

As well as these sets and functions, the following terms can also be used to further describe parts of the abstract syntax.

A *simple phylum* is a phylum such that the cardinality of the set of operators which are members of the phylum is one; that is, the phylum has only one member: $n(\pi(p)) = 1$, where $p \in \mathcal{P}$ and p is a simple phylum and $n(X)$ is the number of members in the set X.

An *optional phylum* is used where a syntactic category need not be expanded. For example, the parameter transmission mode specification for variables passed to Ada procedures is optional. As an operator is required for expansion purposes, the "Null" operator is provided. That is, the operator "Null", which has arity zero, is used to indicate that no syntactic category was chosen.

A *dynamic operator* is an operator that, rather than having a syntactic structure, needs a string of characters to represent it. That is, rather than breaking the operator down any further, a string can be used to represent it; examples include replacing an identifier placeholder with the name of the identifier, or replacing a number operator with the string representing some number. Thus $\sigma(\mathcal{D}) = \phi$, the empty set, where $\mathcal{D}$ is a dynamic operator; note also that $\mathcal{D} \subseteq \mathcal{F}$.

The *starting phylum* is the phylum which represents the root symbol, which is expanded to eventually build a unit to be manipulated.

2.2 Example

To make these definitions clearer, a short example using the syntactic category *statement* from the language Ada follows:

Phyla

Simple_Statement_list, Simple_Case_list, Simple_Else_statements, Expression, Statement, Simple_Case_alternative.

Fixed Arity Operators

While_loop, Procedure_call, Entry_call, Assignment, Return, Case, . . .

Variable Arity Operators

Statement_list, Case_list.

So applying some of the functions gives:

π(Statement) = {While_loop, Assignment, Procedure_call, Entry_call,
Return, Case, . . . }
π(Simple_Identifier) = {Identifier}
σ(While_loop) = Expression Simple_Statement_list
σ(Assignment) = Simple_Identifier Expression
π(Simple_Statement_list) = {Statement_list}
σ(Case) = Expression Simple_Case_list Simple_Else_statements
τ(Statement_list) = Statement
τ(Case_list) = Simple_Case_alternative

From the definitions defined we can say that "Simple_Statement_list" is a simple phylum, along with "Simple_Identifier". Also, rather than breaking up the operator "Identifier" any further, that is, specifying that it is made up letters and so forth, we can nominate it to be a dynamic operator.

2.3 Abstract Syntax Trees

The abstract syntax of a language leads to the representation of a program by an abstract syntax tree. Each node of the tree consists of a phylum and an operator. The operator is a member of the set obtained by applying the function π to the phylum. The branches from each node are derived from the ordered tuples of the elements given by the function σ applied to the operator in the node, if it is a fixed arity operator. If the operator in the node is of variable arity, then any number of branches may extend from the node. The phylum in each of the children is obtained by applying the function τ.

The root of the abstract syntax tree will contain the starting phylum. Leaf nodes will be contain unexpanded phyla, that is, no operator has been chosen from the set of possible operators for that phylum; the operator in the node is a variable arity operator and no branches have been built from the node yet; the operator is a fixed arity operator with arity zero, such as the "Null" operator; or the operator is a dynamic operator in which case its name is replaced by the string the user has entered.

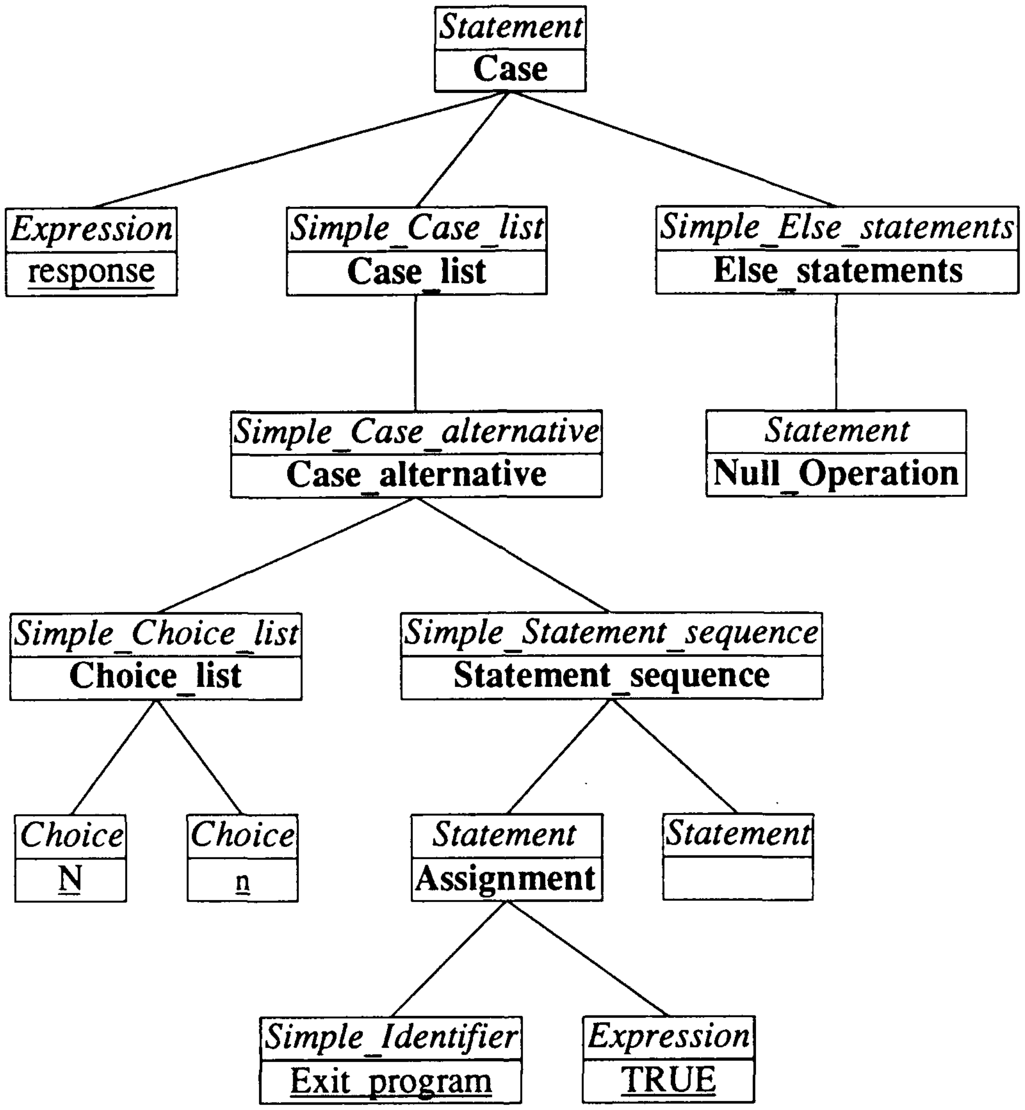


Figure 2 Abstract syntax tree for a *case* statement in Ada.

In Figure 2, an abstract syntax tree is shown for the fragment of Ada program code:

```
case response is
   when 'N' | 'n' => Exit_Program := TRUE;
                     -- No Statement supplied
   when others => null;
end case;
```

From Figure 2, we see that each node consists of a phylum name (in italics) and an operator name (in bold face), The operator *Case* is of fixed arity and it consists of the children in the order defined in the previous example in Section 2.2. The leaves of the tree are either:

Unexpanded nodes (such as the node containing only *Statement*), or
dynamic operators (such as Exit_program — the user has entered the string "Exit_program", which has replaced the dynamic operator **Identifier**), or
fixed arity operators with an arity of zero (such as **Null_Operation**),or
a leaf which is a variable arity operator with no children (such as **Statement_sequence**).

This leads to the method of program creation by expanding the tree from the starting phylum to the desired abstract syntax tree.

3 The Gumnut Meta-Language

The abstract syntax describes a source language mathematically, but it is not suitable for a machine readable description of a programming language. The Gumnut meta-language is an equivalent representation of the abstract syntax. It is described in this section using Extended Backus-Naur Form (EBNF). A concise list of the EBNF given can found in Appendix II.1, whilst Appendix II.2 details the syntax using syntax charts.

3.1 Overall Structure

The overall goal of using this meta-language is to represent a language's abstract syntax definition for use by Gumnut. The main shell of the description is given by

language ::= **language** *language*_name **is**
chapter {chapter} starting_point
end language *language*_name **.**

where *language*_name is the name of the programming language which the abstract syntax is describing. To improve readability within the Meta-language, sections of abstract syntax can be grouped together within a chapter, the syntax for which is

chapter ::= **chapter** *chapter*_name **is**
abstract syntax for *chapter*_name **is**
{definition}
end abstract syntax ;
end chapter *chapter*_name **;**

The *chapter*_name is used to indicate the subject area of the chapter; for example, one may divide an abstract syntax description of Ada into chapters, such as chapters for exception handling, statements and expressions.

3.2 Definitions

The elements of the universe are defined in the chapters, that is, the declarations of phyla and operators. Declarations of phyla and operators may be in any order, so forward references are permitted, but objects should be declared before being referenced, or soon after, to assist human readability. Definitions can be grouped into four separate groups: phylum, optional phylum, operators and dynamic operators.

definition ::= operator_definition | phylum_definition
operator_definition ::= plain_operator | dynamic_definition
phylum_definition ::= optional_definition | ordinary_definition

3.2.1 Phyla

Phylum declarations can be divided into two sections: phylum and optionally null phylum. The only visible difference between the two is the keyword **optional**. The addition of this keyword results in the Null operator being included amongst the sets of each phylum declared with the list of declarations. Other than this, the two types of declarations are identical and so shall be discussed together.

ordinary_definition ::= **phylum** {phylum_list}
phylum_list ::= *phylum*_name : {phylum_struct} ;

phylum_struct ::= *operator*_nam subset_item
subset_item ::= @ *phylum*_name
optional_definition ::= **optional phylum** {phylum_list}

When declaring a phylum, all its members (which are operators) must be specified. To increase readability and simplification, the programmer may place a phylum amongst the operators that are to be included in the phylum's powerset, but only if the phylum is prefixed with an at sign ("@"). This then means that the operators within the powerset of the phylum used are included into the powerset of the phylum being declared. Hence, a phylum may be partitioned into subsets which have different phylum names and are included within the phylum's powerset by using the at sign prefix ("@"). (Using a sub-phylum also leads to some changes in the menus initialization that Gumnut produces, but this will be discussed below.)

```
Name:   Attribute Characters Indexed_comp Slice
        Operator_sym Selected_comp Identifier;
Prefix: Function_Call @Name;
```

In the above declarations, any of the seven defined operators may be used when a name is required. When a prefix is required, either a function call or any of the seven operators defined for name are acceptable for expanding a prefix. Note also that "Name" will be used in operator declarations.

```
Logical_op:         And Or XOr And_Then Or_Else;
Relational_op:      Equal NotEqual Lesser Greater
                    LesserOrEqual GreaterOrEqual
                    In_range NotIn_range;
Binary_adding_op:   Add Subtract Concat;
Unary_adding_op:    UnaryMinus UnaryPlus;
Multiplying_op:     Multiply Divide Modulo Remainder;
Highest_prec_op:    Abs Not Exponent;
Primary_op:         Number NullOperation Aggregate String
                    @Name @Allocator Function_Call
                    Type_convert Qual_expression;
Expression:         @Logical_op @Relational_op @Primary
                    @Unary_adding_op @Binary_adding_op
                    @Multiplying_op @Highest_prec_op;
```

In this example, the phylum Expression has been partitioned into seven sets, which are included in the declaration of Expression. Note, that these seven additional phyla are not used in the declaration of any operators; they are used only to add to the readability of the declaration of expression.

3.2.2 Operators

Operators, by definition, can be either variable or fixed arity. They may also be divided into dynamic operators (the programmer must enter a string to complete the tree), or non dynamic (these are definitions that enable the further expansion of the tree). These two groups of operators are defined separately and distinctly. Variable arity operators and fixed arity operator that are not dynamic, are both declared in a similar fashion, using the structure declared below.

plain_operator ::= **operator** {operator_declaration}
operator_declaration ::= *operator*_name : {fixed_struct | variable_struct} ;

When declaring a fixed arity operator we need to give the details of its ordered tuple. This is achieved by listing, in order, the phyla that are in its tuple. This is done by using the fixed_struct, which is composed of the struct_item, and will be discussed shortly.

fixed_struct ::= {struct_item}

To define a variable arity operator, exactly one phylum is required by definition. The phylum is given via the struct_item, which is the same as that used by the variable arity operator. The number of items of this phylum is given by list_type as either one or more elements ("+"), or zero or more elements ("*"). By allowing both types of lists, the programmer is saved from having to specify a long winded structure. This is really an extension of the rules of abstract syntax.

variable_struct ::= struct_item list_type
list_type ::= + | *

Note that if the list type is not included, then the operator will be assumed to be a fixed arity operator with arity equal to one, because its syntax matches fixed_struct above.

To describe the phyla that appear on the right hand side of the declarations of operators, the struct_item is used. To reference a phylum directly, the name of the phylum is given. An operator may be given when prefixed by a hash ("#"); this can be used when the phylum, that would otherwise have been used, has only the operator in its powerset. This then saves the user from having to declare the phylum, as it is done implicitly. Alternatively, an operator can also be prefixed by a percent sign ("%"); this has the same effect as having used the hash except that the operator is optional. That is, an optionally null phylum is implicitly declared.

struct_item ::= *phylum*_name | simple_phylum | optionally_null_phylum
simple_phylum ::= # *operator*_name
optionally_null_phylum ::= % *operator*_name

Some examples of the declaration of operators is given below. The first of these is:

```
Assignment: Name Expression;
```

which declares the assignment statement to be an operator, which is constructed of a variable name and an expression (that is, the expression is assigned to the variable). Both the expression and variable name are phyla. A constant declaration may be defined by:

```
Constant : #Identifier_list Expression;
```

The operator "Constant" is composed of two fields, an identifier list and an expression. "Identifier_list" is an operator; so it is replaced by the phylum "Simple_Identifier_list", which has only the operator "Identifier_list" in its powerset. That is, the declaration is equivalent to

```
Constant : Simple_Identifier_list Expression;
```

where "Simple_Identifier_list" would be declared elsewhere as a phylum, by:

```
Simple_Identifier_list : Identifier_list;
```

A loop statement could be declared to have three sections, hence:

```
Loop: %Identifier Iteration_Scheme @Statement_Sequence;
```

The first part is a loop identifier which is optional, hence (since it is an optional operator), "%" can be used to simplify the declaration.

```
Statement_sequence: Statement*;
```

A statement sequence is declared to be a list of zero or more statements.

```
Literal_list: Enumeration_literal+;
```

Here a literal list (enumeration list) is declared to have at least one enumeration literal.

A dynamic operator has an arity of zero and hence requires no declaration involving phyla. However the dynamic operator during the construction of the abstract syntax tree requires a string representing that dynamic operator. For example, if the dynamic operator is an identifier, the node requires a string of characters that represents an identifier. This string replaces the identifier operator. To check a string entered as a dynamic operator, a procedure must be supplied that can be called to scan the string entered for any errors. To do this the name of the procedure is declared along with the operator. (The parameters and return value are the same for all procedures, thus allowing a common interface from the tools to those procedures.)

dynamic_definition ::= **dynamic operator** {dynamic_operator}
dynamic_operator ::= *operator_*name **is** *procedure_*name **;**

An example of its use is

```
dynamic operator
  Identifier is ScanIdentifier;
```

Hence when "Identifier" is used, the procedure "ScanIdentifier" will be called to scan the text that is entered by the programmer, after they have been prompted for an identifier.

3.3 Starting Phylum

All that remains to be discussed now is the starting phylum. This is done as the last thing in declaring the abstract syntax for a language. It is done by the following statement:

starting_point ::= **starting phylum** struct_item **;**

"struct_item" is defined in Subsection 3.2.2.

3.4 Semantics

When a name is encountered for the first time, its location determines its type.

Language : declared in the language heading.
Chapter : declared in the chapter heading.
Phylum : declared on the left hand side of both an optional phylum and an ordinary phylum definition; within the definition of the starting phylum; on the right hand side of either of the phylum definitions (if preceded by an at sign ("@")); and on the right hand side of an operator definition.
Operator : declared on the left hand side of an operator definition, (which can be variable, fixed or dynamic); on the right hand side of a phylum definition; and on the right hand side of an operator definition (fixed or variable only); and within the definition of the starting phylum, if prefixed by a hash ("#") or a percent ("%").

The only semantic rule, is that once a name has been declared it must be used consistently. That is, if a name is declared as an operator, it can not later be used as a chapter name. Note, that although procedure name is mentioned during the definition of a dynamic operator, the name given has no semantic meaning, and is only stored with the dynamic operator, to indicate the name of the procedure which will scan input representing the dynamic operator.

3.5 Comments

As in many languages, comments can be added to the abstract syntax description to improve the readability of the description. To add a comment simply enter a hyphen ("–") and any text to the end of the line will be treated as a comment (which is similar to how Ada handles comments).

THIS PAGE INTENTIONALLY LEFT BLANK

4 Gumnut Processing

The Gumnut translator has three distinct stages when it is invoked to translate the description of a language to the modules necessary for generating an instance of MultiView for the described language. If a stage is not completed correctly, that is, an error is found with the language description, then translation stops, and the next stages are not attempted. A data flow diagram, separating out the three stages, is shown in Figure 3 (this is in fact an expansion of the "Run Gumnut translator" bubble, from Figure 1).

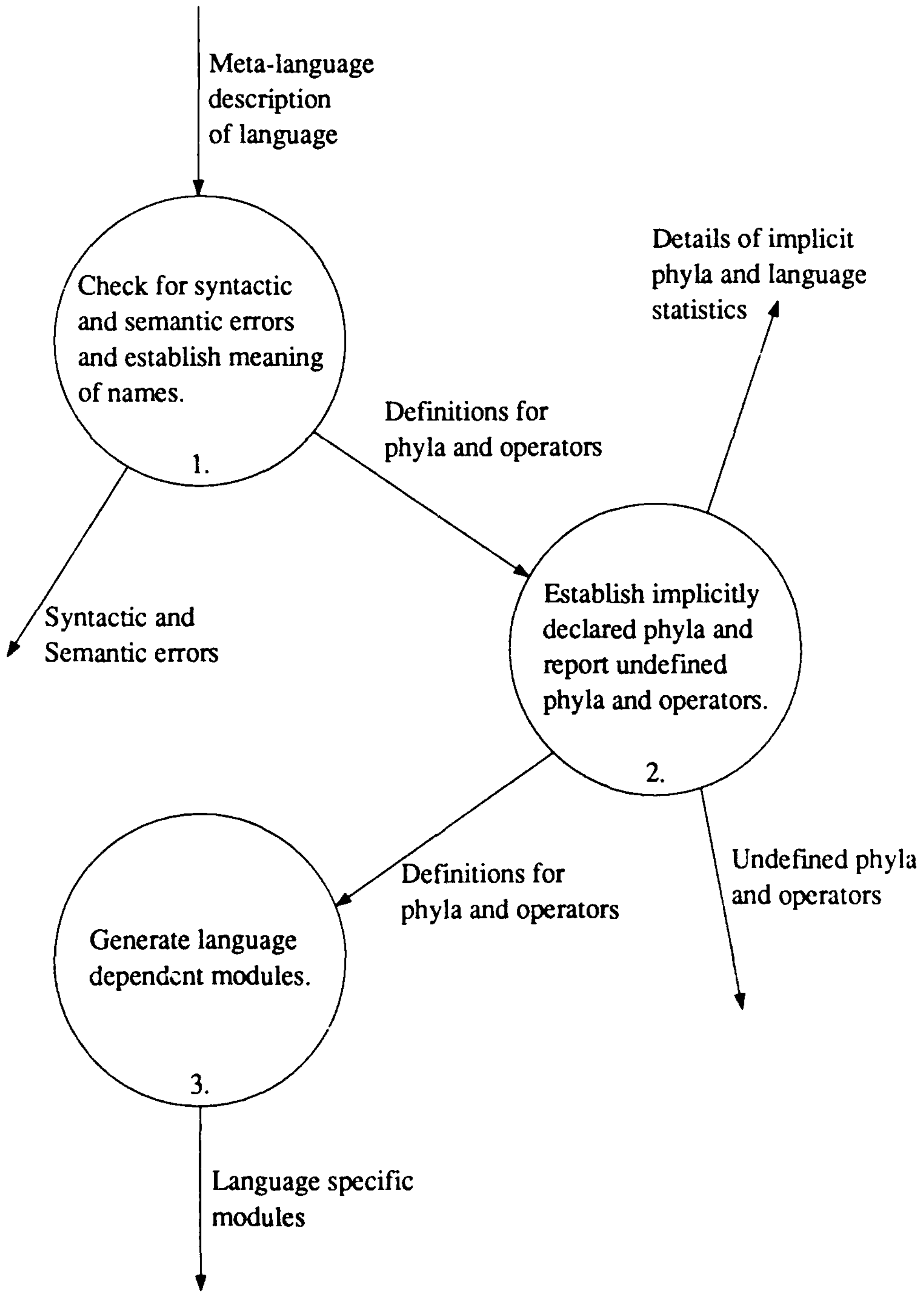

Meta-language description of language
Check for syntactic and semantic errors and establish meaning of names.
1.
Details of implicit phyla and language statistics
Definitions for phyla and operators
Syntactic and Semantic errors
Establish implicitly declared phyla and report undefined phyla and operators.
2.
Definitions for phyla and operators
Undefined phyla and operators
Generate language dependent modules.
3.
Language specific modules

Figure 3 Data Flow Diagram of Gumnut translator.

4.1 First Stage: Syntactic and Semantic Errors

The first stage of translation is to scan the file for syntactic and semantic errors according to the syntax and semantic rules defined in Chapter 3. Error messages are straightforward. Syntactic errors report what is missing and on what line. In the following example, a colon is missing from the line.

```
phylum
  Statement @Compound_Statement @Simple_Statement;
```

This produces the following output from Gumnut (assuming the above description occurs on line 14 and 15 of the input file).

```
Error () Line 15: Colon expected!
```

Note, that the Gumnut compiler does not employ outstanding error recovery, so errors may propagate. For example, in the following code fragment, the declaration of "Statement" is not made within a context of a declaration, such as a phylum.

```
language Incorrect is
  chapter Only is
  abstract syntax for Only is
      - keyword "phylum" has been omitted here.
      Statement: Call;
    dynamic operator
      Ident = Scan;
    operator
      Call : @Ident;
  end abstract syntax;
  end chapter Only;
starting phylum Statement
end language Incorrect.
```

This produces the following errors (from the current version of Gumnut):

```
Error () Line 7: Keyword 'end' expected.
Error () Line 8: Keyword 'abstract' expected.
Error () Line 8: Keyword 'syntax' expected.
Error () Line 8: Keyword 'end' expected.
Error () Line 9: Keyword 'chapter' expected.
Error (Only) Line 9: Chapter name expected.
Error () Line 9: Semi-colon expected.
Error () Line 11: Keyword 'chapter' expected.
Error () Line 11: Keyword 'is' expected.
Error () Line 13: Keyword 'chapter' expected.
Error () Line 14: Expecting an identifier.
Error () Line 14: Keyword 'starting' expected.
Error () Line 14: Keyword 'phylum' expected.
Error () Line 14: Expecting an identifier.
Error () Line 14: Keyword 'language' expected.
Error (Example) Line 14: Language name expected.
Error () Line 15: Dot expected.
```

Semantic errors report the name that is being used and how it is conflicting with what it is thought to be; again the line number is included. For example,

```
Error (Slice) Line 25: Only phyla may appear on right
                       hand side of operator declaration.
```

indicates that "Slice" had previously been declared to be an operator, but is now being used as a phylum within the declaration of an operator on line 25 of the input file.

Also during this process of detecting syntactic and semantic errors, a name, when initially declared, must be stored along with what it represents. When the names are finally defined (or they may be defined immediately), their definitions must be stored with them (for operators and phyla only). For example, when a variable arity operator is defined, the phylum that it may expand to must be stored with the operator along with the list type.

Having successfully completed this stage, a table will have been built up, containing all the information about the operators and phyla defined, as well as those defined implicitly (that is, simple and optionally null phyla).

4.2 Stage Two: Analysis of Names

This stage lists all phyla that have been declared implicitly, either as simple phyla or optional phyla (under the heading "Implicit phyla"). The operators and phyla used but not defined are also reported during this stage of the compilation (under the headings "Unelaborated phyla" and "Unelaborated operators"), or if there are no unelaborated operators or phyla, a message will appear ("No unelaborated phyla." and/or "No unelaborated operators.").

The information that follows supplies some statistics about the declaration of phyla and operators. Under the heading of "Phyla", we see the number of phyla that have been declared explicitly and implicitly, and also the maximum number of operators in any phylum's set. This is followed by two numbers. The first number represents how many operators are in a set, whilst the second number represents how many phyla have that many operators in their set. Under the heading of "Operators", we see the total number of operators, with this figure broken down into the number of fixed and variable arity operators. The highest arity found amongst all operators follows. Again a list of two numbers is printed. The first number represents the arity and the second number represents the number of operators having that arity. The following is the output for the example in Appendix I.

```
Phyla: 11 Max members: 8

         1 5
         2 3
         4 1
         7 1
         8 1
 Operators: 15 Fixed: 13 Variable: 2 Max arity: 2
         0 4
         1 3
         2 6
```

4.3 Stage Three: Production of Files

This stage creates the files: **Language.def, Language.mod, Parsers.def, Parsers.mod**, and **phylum_menus.c**. These files are then compiled and linked to the remainder of the MultiView sources, which are static, and an instance of MultiView is created for the programming language described by the meta-language description. The generation of these files will be discussed in the following chapter.

5 Output From Gumnut

Gumnut produces four Modula-2 source files **Language.def, Language.mod, Scanners.def, Scanners.mod** and a C source file **phylum_menus.c**. The following sections will describe each of the modules. An example can be found in Appendix I.

5.1 Language

Modula-2 source packages comprise a definition and an implementation module, with the UNIX® file extentsions ".def" and ".mod" respectively. **Language.def** contains a type declaration of the Universe as a range, as well as subsequent type declarations of phyla and the various types of operators, all of which are sub-ranges of the Universe. To generate these ranges the following constants need to be determined.

MinFixedArity The start of the fixed arity range, as well as the start of the Universe range. This will always be one.
MaxFixedArity The end of the fixed arity range. This is equal to the number of fixed arity operators including dynamic operators and the operator "NULL".
MinVariableArity The start of the variable arity range. This is **MaxFixedArity** plus one.
MaxVariableArity The end of the variable arity range. Equivalent to the number of variable arity operators plus **MaxFixedArity**. Hence, if there are no variable arity operators used in the abstract syntax description then this number will be one less than **MinVariableArity**.
MinPhylum This is the commencement of the range, used to denote phyla. It is **MaxVariableArity** plus one.
MaxPhylum This represents the conclusion of the phyla range. It is **MaxVariableArity** plus the number of phyla in the abstract syntax definition (both explicitly and implicitly declared). As well as being the end of the phyla range, this is also the end of the Universe range.

Thus the range **MinFixedArity** to **MaxPhylum** enumerates the Universe which is partitioned into fixed arity operators, variable arity operators and phyla.

In addition to constants being required for the declaration for the Universe, the following constants are required for structure declarations, used to store the abstract syntax definition of the language.

MaxArity This number is the maximum arity of all the fixed arity operators. It is used to determine the bounds on an array, which is used to store the phyla of a fixed arity operator.
NoOfOperators This number represents the total number of fixed and variable arity operators and in fact, is equivalent to **MaxVariableArity**. It determines how large a set must be to allow each phylum to have a set of operators, which represents the phylum's powerset of operators. (Due to the limitations of some Modula-2 compilers, the set of operators is broken up into a number of eight member sets.)
NameLength This is the longest identifier used in describing any phyla or operator. Note that this includes the names of implicit phyla that have been generated by Gumnut. The constant is then used to determine the bounds on the character array used for storing each name of the elements of the Universe.

UNIX is a registered trademark of AT & T.

Finally, the starting phylum needs to be defined, so that one can determine the root node of the abstract syntax tree. This is done be declaring the following constant:

StartingPhyla This is the number that has been attributed to the phylum which the starting phylum construct (within the meta-language description of the language) has nominated to be the starting phylum of the language.

Language.mod requires all the initializations necessary to declare the operator sets of each phylum and to define how many fields each operator has, and what those fields are. (The declarations of these structures are found in **Language.def**.) Also needed are the names of the functions that the dynamic operators need to call when text is to be scanned.

Contained in this file are procedures that dynamic operators use to scan text. These procedures are imported from **Scanners**. The initialization of the array **PhylumTable** array is accomplished by:

For each phylum

Name This field contains only the name that was used to declare the phylum.
Members Due to the limitations on set sizes, this set is implemented as an array of eight element sets. The length of the array is from zero to the number of operators div eight (as seen in its declaration within Language.def). When initialized, this array contains the numbers of operators within the phylum's powerset. An operator is added to the array thus:

The set it goes into is given by the number of the operator div eight, and then the set *element it represents is the number of the operator modulo eight.*

For example, an operator which is represented by the number 28, will go into the fourth element (since 27 modulo 8 is 4) of the third set (since 28 div 8 is 3).
Member This is the number used to represent one of the operators in the phylum's set of members.
Simple This is true if the phylum is simple, that is, it has only one operator in its set of members.

The initialization of the array **FixedArityOperatorTable** proceeds as follows:

For each fixed arity operator including the operator NULL which has the value of zero

Name This field just contains the name that was used to declare the phyla.
Arity The number of phyla making up the ordered set of tuples of the operator.
Dynamic This is true if the operator is a dynamic operator, otherwise it is false.
ProcToCall This field can only be elaborated if Dynamic is true, in which case the field contains the procedure to call to scan text that has been entered to represent the dynamic operator.

Finally, the **VariableArityOperator** array is initialized. It proceeds as follows:

For each variable arity operator

Name This field contains only the name that was used to declare the phyla.
AtLeastOne This is true if the operator must have at least one branch containing the phylum given by Operand.
Operand This contains the number of the phylum which the operator can expand to, any number of times.

5.2 Scanner

Both files comprising Scanners, that is, **Scanners.def** and **Scanners.mod**, require the names of the procedures that will be called for scanning text representing dynamic operators. Once these names are known, both the files can be generated.

Note that the bodies of the scanning functions need to be completed by the programmer (in **Scanners.mod**), the error types need to be defined (in **Scanners.def**), and the error messages for each error type need to be written within the procedure SyntacticErrorMessage (in **Scanners.mod**). The completion of the bodies will be discussed in Chapter 6.

5.3 Menus

Template driven editors within the MultiView environment need to be able to display menus such as those shown in Figure 4. Gumnut produces a set of tables used to create such menus, in the file **phylum_menus.c**. To construct this file Gumnut must generate the size of the array used to store the menus used by the textual editor for expanding phyla, and also the text that will generate the menus which are stored in the phylum menu array (in elements according to the number used to represent each phylum). The remainder of the file is static. To define the bounds of the array, the constants **min_phylum**, **max_phylum** and **no_of_phylum**, which correspond to the constants declared in **Language.def** (with the exception that **no_of_phylum** is evaluated). To generate the elements of this array, **ph_menu**, the following algorithm is used.

For each phylum, the corresponding array element is created using "menu_create" with the fields set as follows:

MENU_TITLE_ITEM This is the name used to represent the phylum. It is the same name as that stored for the phylum in the PhylumTable array.

Then, for each item on the right hand side of the phylum's declaration, a **MENU_ITEM** is created thus:

MENU_STRING The name used to represent the operator.

If the item was an operator then **MENU_VALUE** is set thus:

MENU_VALUE The number used to refer to the operator.

Otherwise, the item must have been a phylum, and so a pullright menu is created thus:

MENU_PULLRIGHT The phylum's number is used to reference this array and point to the menu declared for the phylum.

As seen in the above algorithm, the menus will indicate whether the declaration of the language has partitioned phyla into sub-phyla. A pullright item in the menu has an arrow next to it. Selecting the item and moving right (whilst holding the mouse button down) will result in another menu being displayed, as in Figure 5.

For example, consider the following phyla declarations:

```
Array_type_defn: Constrained_array Unconstrained_array;
Type_definition: Access @Array_type_defn Derived_type
                 Enumeration Integer_type Real_fixed_type
                 Real_floating_pt Record;
```

The menu for the array type definition phylum will contain just the two entries: constrained and unconstrained array, both of which are operators, as seen in Figure 4. In the case of the type definition in Figure 5, the item "Array_type_defn" is seen to be a pullright menu, with the same menu as in Figure 4.

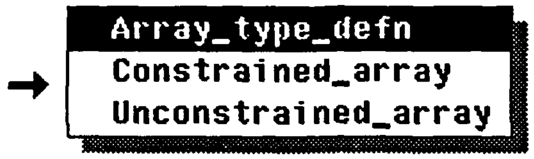


Figure 4 Example of menu used for expanding an array type definition.

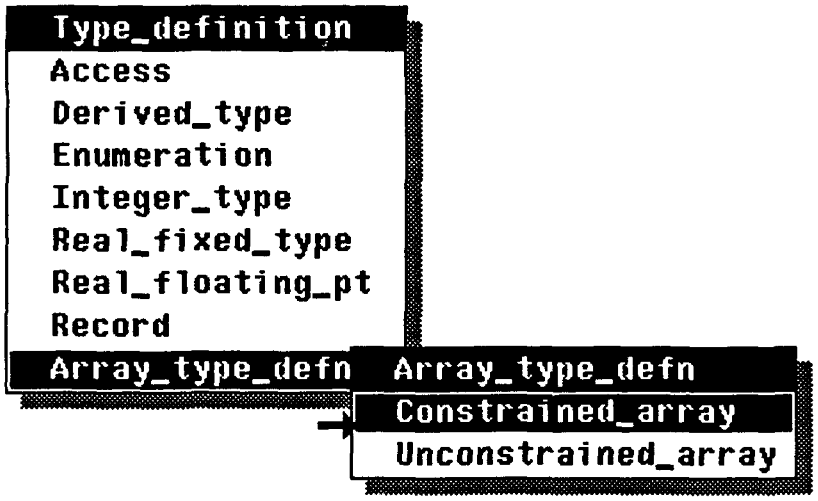


Figure 5 Example of menu used for expanding a type definition phylum.

5.4 Using MultiView

With these modules the MultiView programming environment for the defined language can then be built. Details of this process will given be in later technical reports.

6 Scanners For Dynamic Operators

This chapter describes how the scanners of dynamic operators are developed, how the strings representing dynamic operators are obtained (in general) and the method used to report errors back to the MultiView user. The scanners that have been developed for Ada are described in Appendix III.

6.1 Generation Of Skeleton Procedures

6.1.1 Procedure Stub

As mentioned in Section 3.2.2 a dynamic operator is declared by:

dynamic_operator ::= *operator*_name **is** *procedure*_name ;

The *procedure*_name is then taken to be the name of the procedure for parsing the dynamic operator. Gumnut outputs a module in the Modula-2 language containing stubs for the procedures named in the abstract syntax description. The stubs that are produced in **Scanners.mod** are of the form:

```
PROCEDURE procedure_name (VAR Text : ARRAY OF CHAR)
                              : SyntacticError;
BEGIN
  ;
END procedure_name;
```

which must then be completed by the programmer. The variable "Text" contains the string (an array of characters) which has been entered by the user in response to being asked to enter the dynamic operator. The result is any of the enumerations of "SyntacticError", described in Section 6.1.2 below. If the string entered to represent a dynamic operator is correct then "Success" is returned as the result of the procedure. Otherwise, another enumeration item is returned to indicate the type of error. In developing the scanners for the Ada language, as soon as an error was detected, the error condition was returned as a result of the function; that is, only the first error in the string is detected.

6.1.2 Syntactic Errors

The enumeration items that represent errors found by the scanners are all defined by the programmer, in the file **Scanners.def**. They are declared by adding the enumeration items into the declaration for "SyntacticError", which exists already and contains "Success", as shown here:

```
TYPE
  SyntacticError = (Success);
  (*Add enumerations for errors in above declaration.*);
```

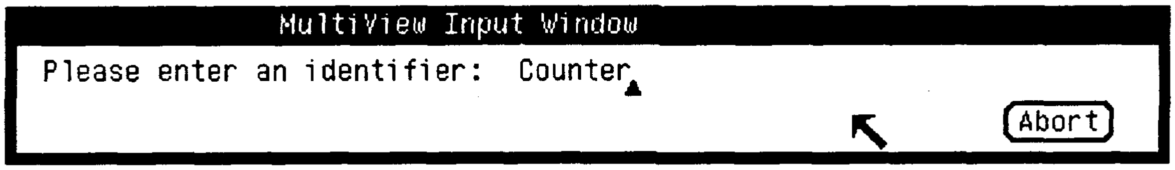
As well as using and declaring the enumeration items that represent errors returned by the dynamic operator scanners, the procedure "SyntacticErrorMessage" must also be extended (it is in the same file as the procedure stubs) to return a string which will then relay the error to the user. For example, an error returned may be "NotDigit" meaning that a digit was expected in the string but not found. Then within the procedure, the following line would be added after the line for returning the message for "Success":

```
| NotDigit: Assign("A digit was expected!",
                   Message, bool);
```

The variable "Message" returns the string to a routine that will then display the message, and "bool" is included to determine if the Assign was successful or not, but will normally be "true" unless the message is more than one hundred characters.

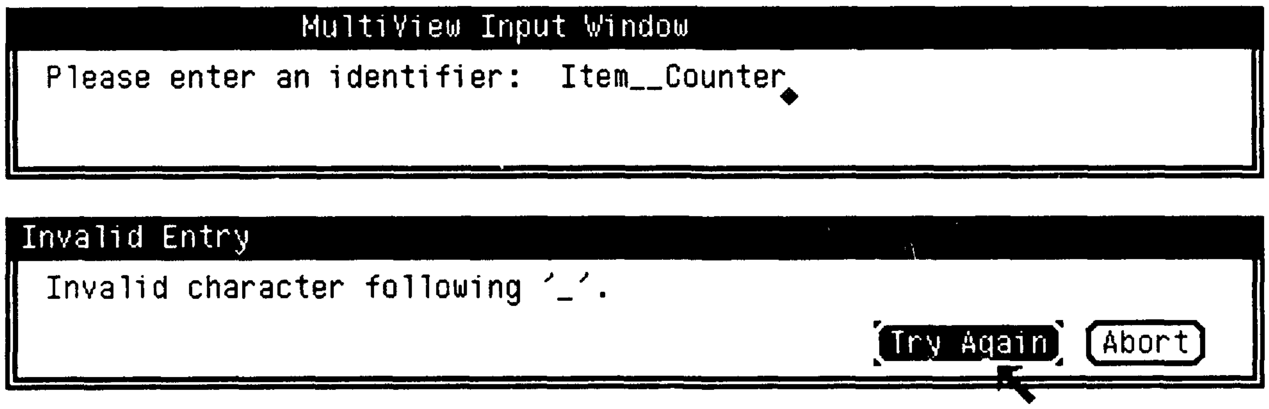
6.2 Interfacing to Template Driven Editors

When a dynamic operator is required a window will appear, such as the one in Figure 6 (assuming the dynamic operator is an identifier). The user of the editor would then enter a string. This string would then be passed to the appropriate scanner routine for scanning. If the string was an acceptable representation of an identifier, then the string would appear in the appropriate location of the identifier in the displayed program segment. If an error was found, an additional window, such as the one in Figure 7, would also appear. We can see that it contains the appropriate error message (obtained by calling SyntacticErrorMessage), as well as a button for the user to try to enter the identifier again, or to abort the entering of the identifier.


MultiView Input Window

Please enter an identifier: Counter

Figure 6 Window containing prompt for an identifier.


MultiView Input Window

Please enter an identifier: Item__Counter

Invalid Entry

Invalid character following '_'.

Figure 7 Window containing details of error in previously entered identifier as well as the window containing the entered identifier.

7 Proposed Extensions

There are still many extensions that can be made to Gumnut to produce more information needed by MultiView to remove all language dependencies from modules not generated by Gumnut.

7.1 Textual Formatting

The textual view, Koala [6], still requires rules to be entered manually to inform the view on how to convert an abstract syntax tree into textual form and also how to indent the text. The current formatting rules used by Koala are those used by the Gandalf system [9].

The formatting rules could be incorporated into the abstract syntax, within the declaration of operators. (Remembering that operators represent a syntactic structure while phyla only represent a syntactic category.) This would mean only changing the meta-language productions:

operator_declaration ::= *operator*_name : {fixed_struct | variable_struct} ;

to the new production:

operator_declaration ::= *operator*_name : [format_rule]
{fixed_struct [format_rule] | variable_struct [format_rule]} ;

where "format_rule" is a set of rules for formatting text and adding keywords; and the meta-language rule for dynamic operators, currently:

dynamic_operator ::= *operator*_name = *procedure*_name ;

to the production:

dynamic_operator ::= *operator*_name = [format_rule] *procedure*_name
*formatting*_name[format_rule] ;

where the procedure given by "*formatting*_name" is used to change the string if necessary. For example, the formatter for strings in Ada will need to add an extra pair of quotes (") when quotes are used within the string.

7.2 Specification of File Extensions

We are developing the capability to specify the file name extension of files that can be read in as textual forms of a program, and also some means of specifying the compilers that can be used. Both of these are needed to remove a few small language dependencies currently existing in modules that are meant to be language independent. We expect that to add these capabilities to Gumnut, it will just require an extension to the meta-language following the declaration of the starting phylum.

These extensions will not result in any change in the specification of the meta-language used to describe a language (that is, within the chapters), but they will just be additional constructs added after the starting point. These constructs will add to the knowledge needed by various modules of MultiView and its views.

7.3 Outputting Files in Other Languages

An extension, which will shortly be incorporated, is the ability to specify the language in which the Language and Scanner modules are output. This choice will be a run-time argument to the

Gumnut command. Ada will be the next language choice available. Thus the command

```
    Gumnut Ada file-name
```

will give Ada code; or

```
    Gumnut Modula-2 file-name
```

will give Modula-2 code, as specified in this document.

7.4 Automatic Creation of Scanners

At present the user has to develop the scanners for the dynamic operators. In the future, the scanners could be generated directly from BNF productions. The technique would be similar to that used by compiler-compilers [1]. The specifications for the scanners might appear with the declaration of the dynamic operators and would replace the naming of the scanning routine.

7.5 Semantic Extensions

There will be more extensions required when MultiView is capable of incremental semantic analysis and code generation. These extensions will allow for the semantic definitions of a language and code generation to be specified, but the requirements are not yet clear.

References

[1] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers; Principles, Techniques and Tools*. Addison Wesley Publishing Company, Reading, Massachusetts, 1986.

[2] R.A. Altmann. An Abstract Syntax Tree Editor for The MultiView Programming Environment. Honours thesis, Department of Computer Science, The University of Adelaide, Adelaide, South Australia, October 1986.

[3] R.A. Altmann, A.N. Hawke, and C.D. Marlin. An Integrated Programming Environment Based on Multiple Concurrent Views. *The Australian Computer Journal*, 20(2):65–72, May 1988.

[4] G. Booch. *Software Engineering With Ada*. The Benjamin/Cummings Publishing Company Inc., Menlo Park, 1983.

[5] Department of Defense, United States of America. *ANSI/MIL-STD-1815A-1983, Ada Programming Language*, February 1983.

[6] C. Lee. A Textual Editor for The MultiView Programming Environment. Honours thesis, Department of Computer Science, The University of Adelaide, Adelaide, South Australia, November 1987.

[7] C.D. Marlin. Multiview: An Integrated Incremental Programming Environment with Multiple Concurrent Views. In *Proc. Seminar on Parallel Computing Architectures*, pages 171–180, Clayton, Victoria, February 1986. Telecom Research Laboratories.

[8] M.J. McCarthy. Towards an Integrated Incremental Programming Environment Based on Multiple Concurrent Processes. Honours thesis, Department of Computer Science, The University of Adelaide, Adelaide, South Australia, October 1985.

[9] B.J. Staudt, C.W. Krueger, A.N. Habermann, and V. Ambriola. The Gandalf System Reference Manuals. Technical report, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, May 1986.

THIS PAGE INTENTIONALLY LEFT BLANK

Appendix I Production of Files and Example

The following BNF is for a simple language, called "Example":

Compilation ::= **program** [Identifier] [Priority] **is** Statements **end**
Statements ::= Statement {; Statement}
Statement ::= Assignment | WriteLn
Assignment ::= Identifier := Expression
WriteLn ::= **put** ([Identifier {, Identifier }])
Priority ::= Expression;
Expression ::= **cosine** Number
| **sine** Number
| Number **plus** Number
| Number **minus** Number
| Number **multiply** Number
| Number **divide** Number
| Number
Identifier ::= letter {letter}
Number ::= [+ | −] digit {digit} [. digit {digit}]

These syntax rules can be translated into the abstract syntax form below, using the Gumnut meta-language.

```
language Example is

  chapter Only is
    abstract syntax for Only is

    dynamic operator
      Number is ScanNumber;

    operator
      Plus     : #Number #Number;
      Minus    : #Number #Number;
      Multiply : #Number #Number;
      Divide   : #Number #Number;
      Cosine   : #Number;
      Sine     : #Number;


    phylum
      BinaryOperations : Plus Minus Multiply Divide;
      TrigFunctions    : Cosine Sine;
      Expression       : Number @BinaryOperations
                         @TrigFunctions;

    optional phylum
      Priority : @Expression;
    operator
      IdentifierList : #Identifier*;
      WriteLn        : #Identifierlist;
      Assignment     : #Identifier Expression;

    phylum
      Statement : Assignment WriteLn;


    dynamic operator
      Identifier is ScanIdentifier;

    operator
      StatementList : Statement+;
      Compilation   : %Identifier #StatementList;

    end abstract syntax;
  end chapter Only;


starting phylum #Compilation
end language Example.
```

To invoke Gumnut, the above meta-language description would appear in a file which has the extension ".ast" (although the extension is not really necessary). Gumnut is then invoked with the command

```
gumnut file.ast
```

where "file.ast" is the name of the file containing the Gumnut meta-language definition of the language for which an instance of MultiView is to be generated. Gumnut then commences translating the file. When this abstract syntax is translated by the Gumnut compiler the resultant files are produced, as specified in Chapter 5:

Language.def

```
(* Language definition module *)

(* This module is used by the Abstract Syntax module *)
(* to contain language information. It should NEVER *)
(* be directly accessed by applications. *)


DEFINITION MODULE Language;

  FROM SYSTEM IMPORT ADDRESS;


  FROM Scanners IMPORT SyntacticError;
  EXPORT QUALIFIED
    LanUniverse, LanOperators,
    LanFixedArityOperators, LanVariableArityOperators,
    LanPhyla, MinFixedArity, MaxFixedArity,
    MinVariableArity, MaxVariableArity,
    MinPhylum, MaxPhylum, StartingPhyla,
    LanObjectNames,
    FixedArityOperatorData, VariableArityOperatorData,
    FixedArityOperatorTable, VariableArityOperatorTable,
    PhylumData, PhylumTable;

  (* Automatically generated constants go here *)

  CONST
    MinFixedArity = 1;
    MaxFixedArity = 13;
    MinVariableArity = 14;
    MaxVariableArity = 15;
    MinPhylum = 16;
    MaxPhylum = 26; MaxArity = 2;
    NoOfOperators = 15;
    StartingPhyla = 26;
    NameLength = 20;

  (* Universe types *)

  TYPE
    LanUniverse  = [MinFixedArity..MaxPhylum];
    LanOperators = [MinFixedArity..MaxVariableArity];
    LanFixedArityOperators
                 = [MinFixedArity..MaxFixedArity];
    LanVariableArityOperators
                 = [MinVariableArity..MaxVariableArity];
    LanPhyla     = [MinPhylum..MaxPhylum];

(* Language data types *)

  TYPE
    zerotomaxarity = [0..MaxArity];
    lenname        = [0..NameLength];
    LanObjectNames = ARRAY [0..NameLength] OF CHAR;
    OperandArray   = ARRAY zerotomaxarity OF LanPhyla;
    SingleParameterFunction
                   = PROCEDURE(VAR ARRAY OF CHAR)
                                      : SyntacticError;

    FixedArityOperatorData =
      RECORD
        Name: LanObjectNames;
        Arity: zerotomaxarity;
        Operand: OperandArray;
        CASE Dynamic: BOOLEAN OF
          TRUE : ProcToCall : SingleParameterFunction;
          END (*CASE*)
        END;

    VariableArityOperatorData =
      RECORD
        Name: LanObjectNames;
        Operand: LanPhyla;
        AtLeastOne: BOOLEAN
        END;

    PhylumData =
      RECORD
        Name:    LanObjectNames;
        Simple:  BOOLEAN;
        Member:  LanOperators;
        Members: ARRAY [0..NoOfOperators DIV 8] OF BITSET
        END;

  VAR
    FixedArityOperatorTable:
      ARRAY LanFixedArityOperators
           OF FixedArityOperatorData;
    VariableArityOperatorTable:
      ARRAY LanVariableArityOperators
           OF VariableArityOperatorData;
    PhylumTable:
      ARRAY LanPhyla
           OF PhylumData;
END Language.
```

<u>Language.mod</u>

```
(* Eucalypt Language Specific Module *)

IMPLEMENTATION MODULE Language;

  FROM Scanners IMPORT

    ScanNumber,
    ScanIdentifier;


  BEGIN
  (* Generated initialization code goes here *)

  WITH PhylumTable [16] DO
    Name := "SimpleNumber";
    Member := 2;     (* Number *)
    Members [0] := {2};
    Members [1] := {};
    Simple := TRUE;
    END;
  WITH PhylumTable [17] DO
    Name := "BinaryOperations";
    Member := 3;     (* Plus *)
    Members [0] := {3,4,5,6};
    Members [1] := {};
    Simple := FALSE;
    END;
  WITH PhylumTable [18] DO
    Name := "TrigFunctions";
    Member := 7;     (* Cosine *)
    Members [0] := {7};
    Members [1] := {0};
    Simple := FALSE;
    END;
  WITH PhylumTable [19] DO
    Name := "Expression";
    Member := 2;     (* Number *)
    Members [0] := {2,3,4,5,6,7};
    Members [1] := {0};
    Simple := FALSE;
    END;
  WITH PhylumTable [20] DO
    Name := "Priority";
    Member := 1;     (* NULL *)
    Members [0] := {1,2,3,4,5,6,7};
    Members [1] := {0};
    Simple := FALSE;
    END;
  WITH PhylumTable [21] DO
    Name := "SimpleIdentifier";
    Member := 9;     (* Identifier *)
    Members [0] := {};
    Members [1] := {1};
    Simple := TRUE;
    END;
  WITH PhylumTable [22] DO
    Name := "SimpleIdentifierlist";
    Member := 11;     (* Identifierlist *)
    Members [0] := {};
    Members [1] := {3};
    Simple := TRUE;
    END;
  WITH PhylumTable [23] DO
    Name := "Statement";
    Member := 12;     (* Assignment *)
    Members [0] := {};
    Members [1] := {2,4};
    Simple := FALSE;
    END;
  WITH PhylumTable [24] DO
    Name := "OptionalIdentifier";
    Member := 9;    (* Identifier *)
    Members [0] := {1};
    Members [1] := {1};
    Simple := FALSE;
    END;
  WITH PhylumTable [25] DO
    Name := "SimpleStatementList";
    Member := 15;    (* StatementList *)
    Members [0] := {};
    Members [1] := {7};
    Simple := TRUE;
    END;
  WITH PhylumTable [26] DO
    Name := "SimpleCompilation";
    Member := 13;    (* Compilation *)
    Members [0] := {};
    Members [1] := {5};
    Simple := TRUE;
    END;
  WITH FixedArityOperatorTable [1] DO
    Name := "NULL";
    Arity := 0;
    Dynamic := FALSE;
    END;
  WITH FixedArityOperatorTable [2] DO
    Name := "Number";
    Arity := 0;
    Dynamic := TRUE;
    ProcToCall := ScanNumber;
    END;
  WITH FixedArityOperatorTable [3] DO
    Name := "Plus";
    Arity := 2;
    Dynamic := FALSE;
    Operand [1] := 16;    (* SimpleNumber *)
    Operand [2] := 16;    (* SimpleNumber *)
    END;
  WITH FixedArityOperatorTable [4] DO
    Name := "Minus";
    Arity := 2;
    Dynamic := FALSE;
    Operand [1] := 16;    (* SimpleNumber *)
    Operand [2] := 16;    (* SimpleNumber *)
    END;
```

```
WITH FixedArityOperatorTable [5] DO
  Name := "Multiply";
  Arity := 2;
  Dynamic := FALSE;
  Operand [1] := 16;     (* SimpleNumber *)
  Operand [2] := 16;     (* SimpleNumber *)
  END;
WITH FixedArityOperatorTable [6] DO
  Name := "Divide";
  Arity := 2;
  Dynamic := FALSE;
  Operand [1] := 16;     (* SimpleNumber *)
  Operand [2] := 16;     (* SimpleNumber *)
  END;
WITH FixedArityOperatorTable [7] DO
  Name := "Cosine";
  Arity := 1;
  Dynamic := FALSE;
  Operand [1] := 16;     (* SimpleNumber *)
  END;
WITH FixedArityOperatorTable [8] DO
  Name := "Sine";
  Arity := 1;
  Dynamic := FALSE;
  Operand [1] := 16;     (* SimpleNumber *)
  END;
WITH FixedArityOperatorTable [9] DO
  Name := "Identifier";
  Arity := 0;
  Dynamic := TRUE;
  ProcToCall := ScanIdentifier;
  END;
WITH FixedArityOperatorTable [10] DO
  Name := "WriteLn";
  Arity := 1;
  Dynamic := FALSE;
  Operand [1] := 22;     (* SimpleIdentifierlist *)
  END;
WITH FixedArityOperatorTable [11] DO
  Name := "Identifierlist";
  Arity := 0;
  Dynamic := FALSE;
  END;
WITH FixedArityOperatorTable [12] DO
  Name := "Assignment";
  Arity := 2;
  Dynamic := FALSE;
  Operand [1] := 21;     (* SimpleIdentifier *)
  Operand [2] := 19;     (* Expression *)
  END;
  WITH FixedArityOperatorTable [13] DO
    Name := "Compilation";
    Arity := 2;
    Dynamic := FALSE;
    Operand [1] := 24;    (* OptionalIdentifier *)
    Operand [2] := 25;    (* SimpleStatementList *)
    END;
  WITH VariableArityOperatorTable [14] DO
    Name := "IdentifierList";
    AtLeastOne := FALSE;
    Operand := 21;       (* SimpleIdentifier *)
    END;
  WITH VariableArityOperatorTable [15] DO
    Name := "StatementList";
    AtLeastOne := TRUE;
    Operand := 23;       (* Statement *)
    END;

  END Language.
```

<u>Scanners.def</u>

```
DEFINITION MODULE Scanners;

  EXPORT QUALIFIED
    ScanNumber,
    ScanIdentifier,
    SyntacticError,
    SyntacticErrorMessage;


  TYPE
    SyntacticError = (Success);
    (*Add enumerations for errors in above declaration.*)

  PROCEDURE SyntacticErrorMessage(Error : SyntacticError;
                                  VAR Mess : ARRAY OF CHAR);

  PROCEDURE ScanNumber (VAR Text : ARRAY OF CHAR)
                          : SyntacticError;

  PROCEDURE ScanIdentifier (VAR Text : ARRAY OF CHAR)
                              : SyntacticError;

END Scanners.
```

Scanners.mod

```
IMPLEMENTATION MODULE Scanners;

FROM String IMPORT Assign;

  PROCEDURE SyntacticErrorMessage
                              (Error : SyntacticError;
                               VAR Message : ARRAY OF CHAR);
  VAR bool : BOOLEAN;
  BEGIN
    CASE Error OF
      Success : Assign("Success",Message,bool);
    ELSE
      ;
    END (*CASE*);
  END SyntacticErrorMessage;

  PROCEDURE ScanNumber (VAR Text : ARRAY OF CHAR)
                            : SyntacticError;
  BEGIN
    ;
  END ScanNumber;

  PROCEDURE ScanIdentifier (VAR Text : ARRAY OF CHAR)
                                : SyntacticError;
  BEGIN
    ;
  END ScanIdentifier;

END Scanners.
```

phylum_menus.c

```
#include <suntool/sunview.h>
#include <suntool/canvas.h>
/* Include necessary header files for attaching the
   menus to the frame used by the view. */
#define min_phylum (16)
#define max_phylum (26)
#define no_of_phyla (11)
#define ph_menu(ph) phylum_menu[ph - min_phylum]

Menu phylum_menu[no_of_phyla];

void init_phylum_menus()
  {
  ph_menu(16) = menu_create (
    MENU_TITLE_ITEM, "SimpleNumber",
    MENU_ITEM, MENU_STRING, "Number", MENU_VALUE, 2, 0,
                                0);
```

```
ph_menu(17) = menu_create (
  MENU_TITLE_ITEM, "BinaryOperations",
  MENU_ITEM, MENU_STRING, "Plus", MENU_VALUE, 3, 0,
  MENU_ITEM, MENU_STRING, "Minus", MENU_VALUE, 4, 0,
  MENU_ITEM, MENU_STRING, "Multiply", MENU_VALUE, 5, 0,
  MENU_ITEM, MENU_STRING, "Divide", MENU_VALUE, 6, 0,
                          0);
ph_menu(18) = menu_create (
  MENU_TITLE_ITEM, "TrigFunctions",
  MENU_ITEM, MENU_STRING, "Cosine", MENU_VALUE, 7, 0,
  MENU_ITEM, MENU_STRING, "Sine", MENU_VALUE, 8, 0,
                          0);
ph_menu(19) = menu_create (
  MENU_TITLE_ITEM, "Expression",
  MENU_ITEM, MENU_STRING, "Number", MENU_VALUE, 2, 0,
  MENU_ITEM,
    MENU_STRING, "BinaryOperations",
    MENU_PULLRIGHT, ph_menu(17), 0,
  MENU_ITEM,
    MENU_STRING, "TrigFunctions",
    MENU_PULLRIGHT, ph_menu(18), 0,
                          0);
ph_menu(20) = menu_create (
  MENU_TITLE_ITEM, "Priority",
  MENU_ITEM, MENU_STRING, "NULL", MENU_VALUE, 1, 0,
  MENU_ITEM,
    MENU_STRING, "Expression",
    MENU_PULLRIGHT, ph_menu(19), 0,
                          0);
ph_menu(21) = menu_create (
  MENU_TITLE_ITEM, "SimpleIdentifier",
  MENU_ITEM, MENU_STRING, "Identifier",MENU_VALUE, 9, 0,
                          0);
ph_menu(22) = menu_create (
  MENU_TITLE_ITEM, "SimpleIdentifierlist",
  MENU_ITEM, MENU_STRING, "Identifierlist", MENU_VALUE, 11, 0,
                          0);
ph_menu(23) = menu_create (
  MENU_TITLE_ITEM, "Statement",
  MENU_ITEM, MENU_STRING, "Assignment", MENU_VALUE, 12, 0,
  MENU_ITEM, MENU_STRING, "WriteLn", MENU_VALUE, 10, 0,
                          0);
ph_menu(24) = menu_create (
  MENU_TITLE_ITEM, "OptionalIdentifier",
  MENU_ITEM, MENU_STRING, "Identifier", MENU_VALUE, 9, 0,
  MENU_ITEM, MENU_STRING, "NULL", MENU_VALUE, 1, 0,
                          0);
ph_menu(25) = menu_create (
  MENU_TITLE_ITEM, "SimpleStatementList",
  MENU_ITEM, MENU_STRING, "StatementList", MENU_VALUE, 15, 0,
                             0);
  ph_menu(26) = menu_create (
    MENU_TITLE_ITEM, "SimpleCompilation",
    MENU_ITEM, MENU_STRING, "Compilation", MENU_VALUE, 1 , 0,
                             0);
  }


unsigned PopPhylumMenu(ph,code)
  unsigned ph, code;
  {
  unsigned store;
  if (code == 1)
    menu_set(ph_menu(ph),
               MENU_INSERT, 1,
               menu_create_item(MENU_STRING, "Comment Out",
                                MENU_VALUE, 1000, 0),
               0);
  if (code == 2)
    menu_set(ph_menu(ph),
               MENU_INSERT, 1,
               menu_create_item(MENU_STRING, "Comment In",
                                MENU_VALUE, 1001, 0),
0);
  store = (unsigned)menu_show(ph_menu(ph),
                              canvas, event, 0);
  if ((code == 1) || (code == 2))
    menu_set(ph_menu(ph),
               MENU_REMOVE, 2, 0);
  return(store);
  }
```

THIS PAGE INTENTIONALLY LEFT BLANK

Appendix II Gumnut Meta Language Description

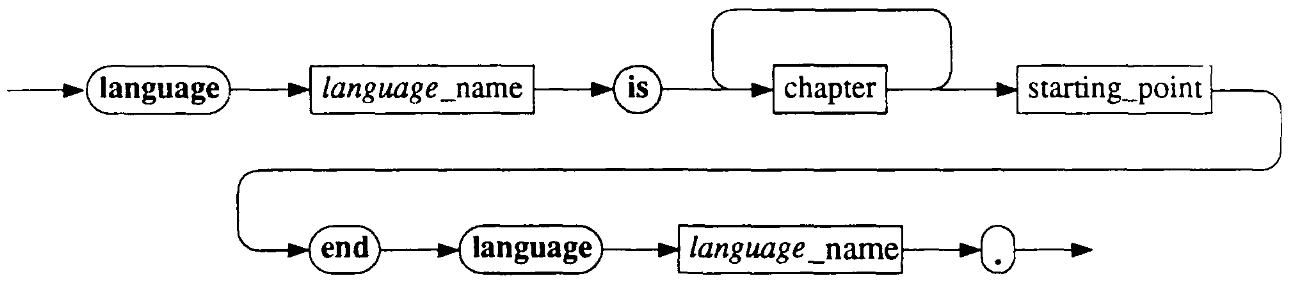
II.1 BNF Description

The BNF rules for the Gumnut meta-language are as follows:

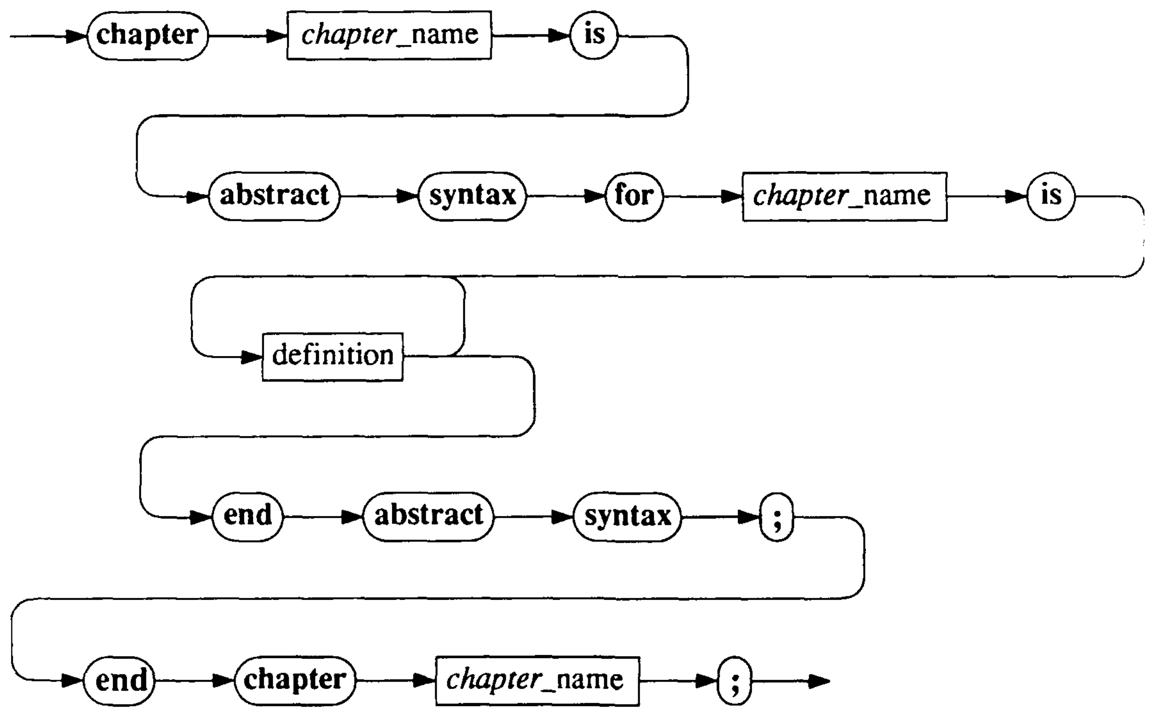
language ::= **language** *language*_name **is**
        chapter {chapter} starting_point
    end language *language*_name **.**
chapter ::= **chapter** *chapter*_name **is**
        abstract syntax for *chapter*_name **is**
            {definition}
        end abstract syntax ;
    end chapter *chapter*_name **;**
definition ::= operator_definition | phylum_definition
operator_definition ::= plain_operator | dynamic_definition
phylum_definition ::= optional_definition | ordinary_definition
plain_operator ::= **operator** {operator_declaration}
operator_declaration ::= *operator*_name **:** {fixed_struct | variable_struct} **;**
fixed_struct ::= {struct_item}
variable_struct ::= struct_item list_type
list_type ::= **+** | *****
struct_item ::= *phylum*_name | simple_phylum | optionally_null_phylum
simple_phylum ::= **#** *operator*_name
optionally_null_phylum ::= **%** *operator*_name
dynamic_definition ::= **dynamic operator** {dynamic_operator}
dynamic_operator ::= *operator*_name **is** *procedure*_name **;**
ordinary_definition ::= **phylum** {phylum_list}
phylum_list ::= *phylum*_name **:** {phylum_struct} **;**
phylum_struct ::= *operator*_name | subset_item
subset_item ::= **@** *phylum*_name
optional_definition ::= **optional phylum** {phylum_list}
starting_point ::= **starting phylum** struct_item **;**
name ::= letter { [underscore] letter}
letter :: A | ... | Z | a | ... | z

II.2 Syntax Charts

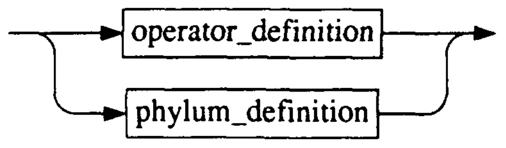
Language



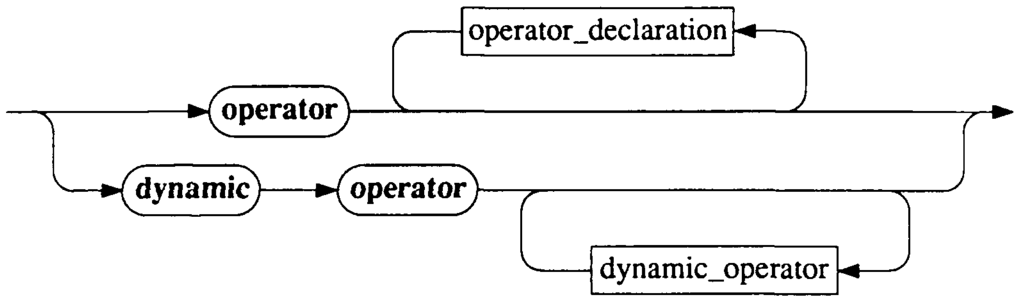
Chapter



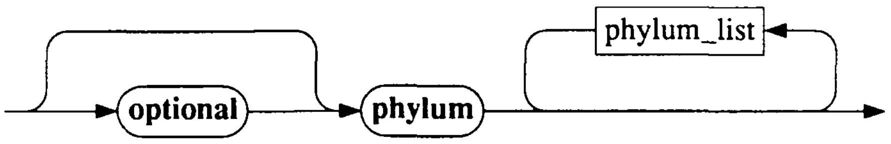
Definition



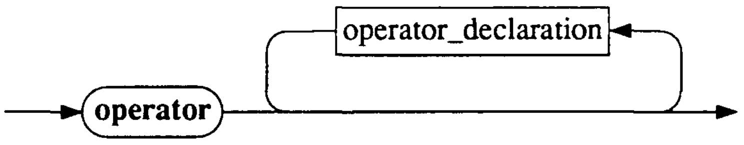
Operator definition



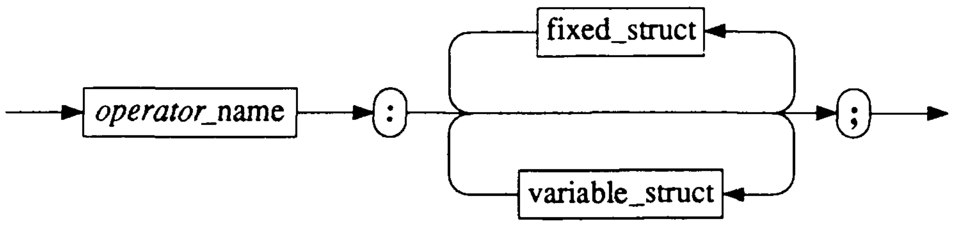
Phylum definition



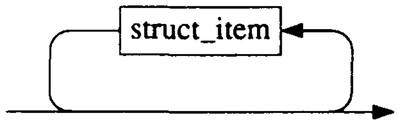
Plain operator



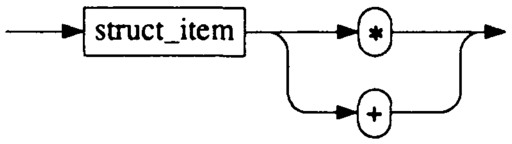
Operator declaration



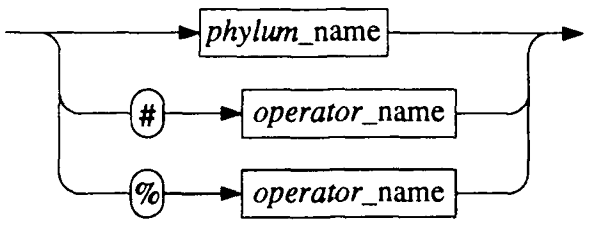
Fixed struct



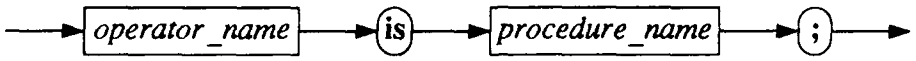
Variable struct



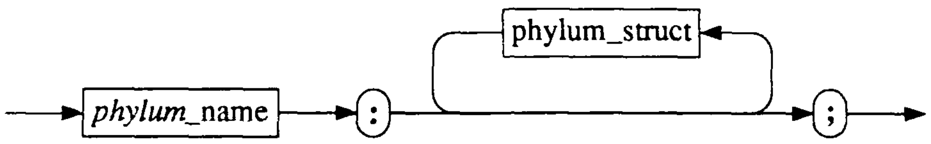
Struct item



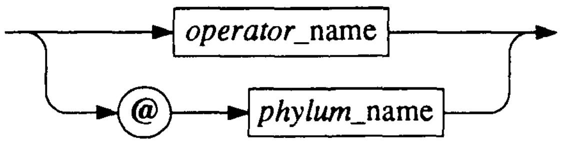
Dynamic operator



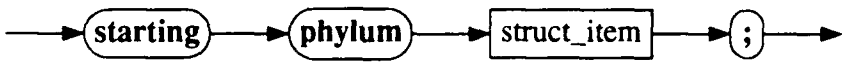
Phylum list



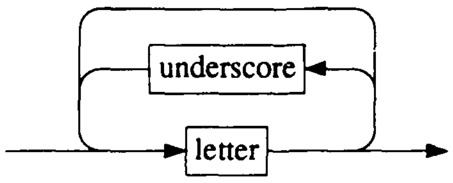
Phylum struct



Starting point



Name



Appendix III Scanners Available

The following table (Table 1) lists the scanners currently implemented for use with MultiView, which has been instantiated for the Ada programming language.

Procedure	BNF	Error Conditions
ScanIdentifier		
identifier	::= letter {[underscore] letter_or_digit}	Zero length
letter_or_digit		Illegal character
	::= letter | digit	Error using underscore
ScanNumber		
number	::= decimal_number | based_number	Zero length
decimal_number		Illegal character
	::= integer [.integer] [exponent]	Error using underscore
integer	::= digit {[underscore] digit}	Error using dot
exponent	::= E [+] integer | E - integer	Error in the exponent
based_number		
	::= base # based_integer [.based_integer] # [exponent]	Missing hash in based number
base	::= integer	
base_integer		
	::= extended_digit {[underscore] extended_digit}	
extended_digit		
	::= digit | A | B | C | D | E | F	
ScanString		
string	::= {character}	Invalid character
ScanChar		
character_literal		No character given
	::= character	Invalid character

Table 1 Table summarizing procedures supplied for scanning Ada lexical items.

Note that although the strict BNF form for character strings in Ada is:

string ::= "{character}"

The quotes (") need not be entered by the MultiView user because syntactic symbols are implicit. Thus, any string can be entered. Also, there is usually some convention for entering the quotes used to denote strings (in the case of Ada two quotes are placed together to represent one quote). However, this is not needed as the environment knows that the string is terminated by other conditions, not a quote. Similarly for the character literal, although the BNF rule is:

character_literal ::= 'character'

The quotes (') are implicit and must not be entered.

DISTRIBUTION

	Copy No
DEPARTMENT OF DEFENCE	
Defence Science and Technology Organisation	
Chief Defence Scientist	)
First Assistant Secretary Science Policy	) 1
Director General Science and Technology Programs	)
Counsellor, Defence Science, London	Cnt Sht Only
Counsellor, Defence Science, Washington	Cnt Sht Only
Defence Science Representative, Bangkok	Cnt Sht Only
Scientific Adviser, Defence Research Centre, Kuala Lumpur	Cnt Sht Only
Electronics Research Laboratory	
Director, Electronics Research Laboratory	2
Chief, Information Technology Division	3
Chief, Communications Division	4
Chief, Electronic Warfare Division	5
Head, Software Engineering Group	6 - 7
Head, Information Systems Research Group	8
Head, Trusted Computer Systems Group	9
Publicity and Component Support Officer, Information and Technology Division	10
Aeronautical Research Laboratory	
Director, Aeronautical Research Laboratory	11
Materials Research Laboratory	
Director, Materials Research Laboratory	12
Surveillance Research Laboratory	
Director, Surveillance Research Laboratory	13
Weapons Systems Research Laboratory	
Director, Weapons Systems Research Laboratory	14
Head, Combat Systems Integration Group	15
Army Office	
Scientific Adviser - Army	16
Director General Army Dvelopment (NSO), Russell Offices for ABCA Standardisation Officers	
UK ABCA Representative	17
US ABCA Representative	18
Canada ABCA Representative	19
New Zealand ABCA Representative	20
Air Office	
Air Force Scientific Adviser	21
Joint Intelligence Organisation (DSTI)	22
Director of Departmental Publications	23

Libraries and Information Services	
Librarian, Technical Reports Centre, Defence Central Library, Campbell Park	24
Document Exchange Centre	
Defence Information Services and Science Liaison Branch for:	
Microfilming	25
United Kingdom, Defence Research Information Centre	26 - 27
United States, Defense Technical Information Center	28 - 39
Canada, Director, Scientific Information Services	40
New Zealand, Ministry of Defence	41
National Library of Australia	42
Main Library, Defence Science and Technology Organisation Salisbury	43 - 44
Australian Defence Force Academy Library	45
Library, Aeronautical Research Laboratories	46
Library, Materials Research Laboratories	47
Librarian, DSD, Melbourne	48
Office of Defence Production	
Chief of Defence Production	49
Defence Industry and Materiel Policy Division	
FASDIMP	50
ACADEMIC INSTITUTIONS	
University of Adelaide, Dr Chris Marlin, Department of Computer Science	51
University of Queensland, Prof A.M. Lister, Department of Computer Science	52
Royal Melbourne Institute of Technology, Mr L. Jackson, Centre for Advanced Telecommunications Technology	53
Australian National University, Prof R.B. Stanton, Department of Computer Science	54
Author	55 - 56
Spares	57 - 62

DOCUMENT CONTROL DATA SHEET

Security classification of this page: UNCLASSIFIED

1 DOCUMENT NUMBERS	
AR Number:	AR-006-402
Series Number:	ERL-0512-TR
Other Numbers:	

2 SECURITY CLASSIFICATION	
a. Complete Document:	Unclassified
b. Title in Isolation:	Unclassified
c. Summary in Isolation:	Unclassified

3 DOWNGRADING/DELIMITING INSTRUCTIONS

N/A

4 TITLE

GUMNUT SPECIFICATION AND REPORT

5 PERSONAL AUTHOR(S)

R.A. ALTMANN,
M.A. FITZGERALD and
P.S. KEAYS

6 DOCUMENT DATE

MAY 1990

7 7.1 TOTAL NUMBER OF PAGES	43
7.2 NUMBER OF REFERENCES	9

8 8.1 CORPORATE AUTHOR(S)

ELECTRONICS RESEARCH LABORATORY

8.2 DOCUMENT SERIES and NUMBER

TECHNICAL REPORT 0512

9 REFERENCE NUMBERS	
a. Task:	N/A
b. Sponsoring Agency:	N/A

10 COST CODE

N/A

11 IMPRINT (Publishing organisation)

DEFENCE SCIENCE AND
TECHNOLOGY ORGANISATION

12 COMPUTER PROGRAM(S)
(Title(s) and language(s))

13 RELEASE LIMITATIONS (of the document)

APPROVED FOR PUBLIC RELEASE

89/49

Security classification of this page: UNCLASSIFIED

Security classification of this page: UNCLASSIFIED

14 ANNOUNCEMENT LIMITATIONS (of the information on these pages)

NO LIMITATION

15 DESCRIPTORS		16 COSATI CODES
a. EJC Thesaurus Terms	PROGRAMMING LANGUAGES TRANSLATOR ROUTINES SOFTWARE TOOLS	1205
b. Non-Thesaurus Terms	META-LANGUAGE	

17 SUMMARY OF ABSTRACT
(if this is security classified, the announcement of this report will be similarly classified)

GUMNUT IS A PART OF MULTIVIEW, AN INTEGRATED PROGRAMMING ENVIRONMENT. BY MEANS OF A NUMBER OF TOOLS OPERATING POSSIBLY CONCURRENTLY, OVER A DISTRIBUTED WORKSTATION NETWORK, MULTIVIEW SUPPORTS THE DEVELOPMENT OF SOFTWARE IN A GROWING NUMBER OF PROGRAMMING LANGUAGES. GUMNUT AND ITS ASSOCIATED META-LANGUAGE (OR LANGUAGE TO DESCRIBE A LANGUAGE) IS THE TOOL WHICH ALLOWS MULTIVIEW TO BE EXTENDED FOR A NEW PROGRAMMING LANGUAGE. THIS REPORT DESCRIBES GUMNUT AND THE META-LANGUAGE.

Security classification of this page: UNCLASSIFIED